浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：范源颢

学　　院：计算机科学与技术学院

　　系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3180103574

指导教师：陈文智

2020 年　12 月 22　日

浙江大学实验报告

课程名称：计算机体系结构 实验类型：综合

实验项目名称：支持中断的流水线 CPU

学生姓名：范源颢 专业：计算机科学技术 学号：3180103574

同组学生姓名：周寒靖 指导老师：陈文智

实验地点：曹光彪西楼-301 实验日期：2020 年 1 月 2 日

一、 实验目的和要求

目的：

1. 理解 CPU 中断的原理和处理中断的流程
2. 理解 CPU 中 cp0 协处理器的功能
3. 掌握支持中断 CPU 的设计方法
4. 掌握支持中断 CPU 的程序验证方法

要求：

1. 设计流水线 CPU，使得它支持 2 条中断指令，包括设计数据通路 datapath，CPU 控制器 CPU controller，以及协处理器 CP0。
2. 用程序验证单周期 CPU，观察程序的执行情况。

二、 实验内容和原理

2.1 CP0 的作用和意义

CP0 是协助 CPU 处理的协处理器，他可以确定 CPU 的工作方式，或者改变操作系统的接口，为了完成自己的工作，CP0 自己会包含一些寄存器。CP0 除了完成 CPU 的配置之外，还可以完成的工作包括：缓存控制，中断、异常的控制，存储管理单元控制，以及其他。

2.2 CP 指令

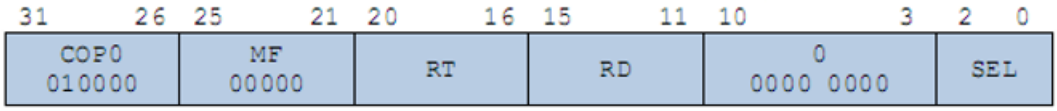
我们这里假定 CP0 也有 32 个 32 位寄存器（事实上 CP0 的寄存器数量一般少于 32 个）。我们这里要求实现的指令有三种，MTC0，MFC0，以及 ERET，他们的字面意思分别是“Move To Cp0”，“Move From Cp0”，“Exception RETurn”。

1. MFC0

MFC0 rt, rd, sel #meaning： GPR[rt] = CPR[0, rd, sel]

就是说，指令 MFC0 是指，将通用寄存器组（General Purpose Registers， GPR）的第 rt 位赋值为协处理器寄存器（CoProcessor Register，CPR）的第 rd 位。在标准 MIPS 语句中，rd 之后会拼接一个 3 比特位的 SEL 来扩展目录空间，在真正实验室，我们会放弃使用 sel,或者把 sel 都设置为 0.

MFC0 的机器码指令格式如下：



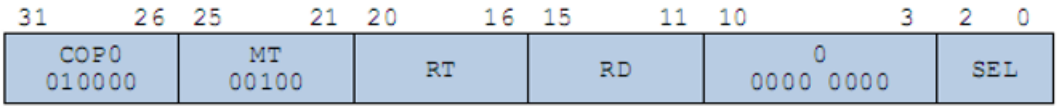
第 31 到 26 位的 010000 指出了他是一条协处理器指令，25 到 21 位的 00000 表示他是 MF 指令，之后两个 5 位的地址就是上文提到的 RT 和 RD，而 10-3 位的 8 个比特位一般置零，最后 3 位的 SEL 作用亦如上文所述。

2. MTC0

MTC0 rt, rd, sel #meaning: CPR[0, rd, sel] = GPR[rt]

和 2.1.1 类似，他的作用是将通用寄存器组的第 rt 位寄存器的值写入到协处理器寄存器中，sel、rt、rd 的意义和 2.1.2 相同，但是我们需要注意的是，结合 MTC 和 MFC，MIPS 汇编码再指令字之后都紧跟 rt, 这是通用寄存器的下标，通用寄存器在线，协处理器寄存器在后。

机器码指令格式如下：



同样适用 010000 表示他是协处理器的指令，MT 的操作码也 5 位，00100.

3. ERET

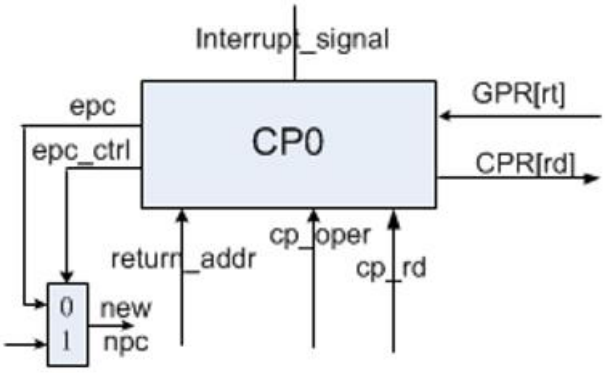
指令格式就是单独的一个指令字

ERET #meaning： jump to a certain address stored in the EPCR

这是一个跳转指令，他必须和中断机制结合在一起解释，一旦中断发生，CPU 需要做的事情有两件事，首先，我们需要记录下发生中断的指令的地址，这样方便我们在处理中断之后返回，这个发生中断的地址一般存储在 CP0 的 EPCR 号位中，EPCR 是指中断程序计数寄存器（Exception Program Counter Register，EPCR）。完成这部之后，我们就可以跳转指令地址到一个专门的处理中断的指令地址段中，这个地址段储存在 CP0 的 EHBR 号位中，义即中断处理基寄存器（Exception Handler Base Register， EHBR，因为中断处理可能需要多次地址跳转，所以这里仅仅称为“基”）。在这些指令地址上运行处理中断的程序，之后程序假如程序需要返回中断的地方继续执行之前发生中断的程序，那么需要调用 ERET，表示中断返回，读取 EPCR 号位中的寄存器的值，作为跳转的目的地址。

2.3 简单中断控制

我们使用下图表示一个协处理器的元件图：



外部的 interrupt_signal 表示产生中断的信号，通过 return_addr，我们可以确定 eret 的地址，因此我们一般将这个信号的值储存在 EPCR 中，中断返回时，CP0 会发出 epc 信号和 epc_ctrl 信号(EPC for Exception Program Counter) ，一个表示中断返回的地址，一个表示需要中断返回，在我们的实验中，我们使用了“jump_addr”和“jump_en”两个信号名。同时，为了支持 MTC0 和 MFC0，我们还需要 CP0 的读写信号，也就是图中的 GPR[rt]和 CPR[rd]（注意 GPR 的索引一定是 rt,而 CPR 的索引一定是 rd）。

2.4 实现机制和一些设计方法

一旦检测到中断，我们需要将在 IF 阶段的指令消灭，同时确定返回地址，这里有一种特殊情况，那就是假如我们执行 IF 的这条指令是一条延时槽当中的指令，那么我们返回的地址应该不是这条指令所在的地址，而应该是这条指令的上一条（一条分支指

令，现在处在 ID 阶段），因为我们在上面的报告里面已经提到，延时槽中的指令是被我们提早读取出来的，因此，延时槽指令实际的下一条指令的地址应该是延时槽指令上一条分支指令的跳转结果，这个机制在实际的程序设计中应该考虑。

另一问题是，我们需要考虑中断不应该被反复进入，我们要考虑，一旦一个时钟周期已经产生了中断信号（interrupt_signal, 我们的实验中成为 ir_in， interrupt_input），那么我们下个时钟周期如果还有中断信号，那么我们的中断信号将不再起作用，否则，一旦中断信号超过了两个时钟周期，我们的协处理器就无法确定到底是那条指令引起的中断了。

为了达到这个效果，我们必须将中断信号储存起来，ir_in 表示中断输入，在每个时钟上升沿，检测到 ir_in 信号，我们就是的 ir_wait 信号置 1（平时为 0），我们再指定另一个信号 ir_valid 信号，通常置 1，并且令 ir = ir_en & ir_wait & ir_valid, ir 表示中断处理开始的信号。这样在上文所述的情况下，我们可以正常进入中断跳转。下一个时钟周期上升沿时，我们一旦发现 ir 为 1，我们就将 ir_valid 置为 0，那么无论下一个时钟周期是否有 ir_in 信号，ir 信号都是 0.因此，就在下一个周期， ir_valid 将会重新变成 1。这样就防止了两个时钟周期的中断信号对中断系统的破坏。

实验给出了参考代码：

```
// interrupt determination
wire ir;
reg ir_wait = 0, ir_valid = 1;
reg eret = 0;
always @(posedge clk) begin
    if (rst)
        ir_wait <= 0;
    else if (ir_in)
        ir_wait <= 1;
    else if (eret)
        ir_wait <= 0;
end
always @(posedge clk) begin
    if (rst)
        ir_valid <= 1;
    else if (eret)
        ir_valid <= 1;
    else if (ir)
        ir_valid <= 0;  // prevent exception reenter
end
```

```
assign ir = ir_en & ir_wait & ir_valid;
```

（注意 ir_en 是和 ir_in 一样的外生输入变量，表示其他部件是否允许 cp0 产生中断）

2.5 FPGA 的抖动控制：

尽管我们上文去除了中断重进入的问题，另外还有一个问题，就是抖动，我们在实验中是通过按压按钮模拟中断的产生的，而一次按压持续的时间可能远大于几次时钟周期，于是依旧会产生了 ir_in 信号，尽管这些 ir_in 信号因为 2.4 设计的原因不会产生中断重进入的问题，但是会产生“多次中断”，也就是按压一次按钮，一连串指令进入中断的情况，这不是程序设计的错误，但确实会使得程序的运行在人机交互上令人觉得怪异，因此我们需要加上消除抖动的逻辑。

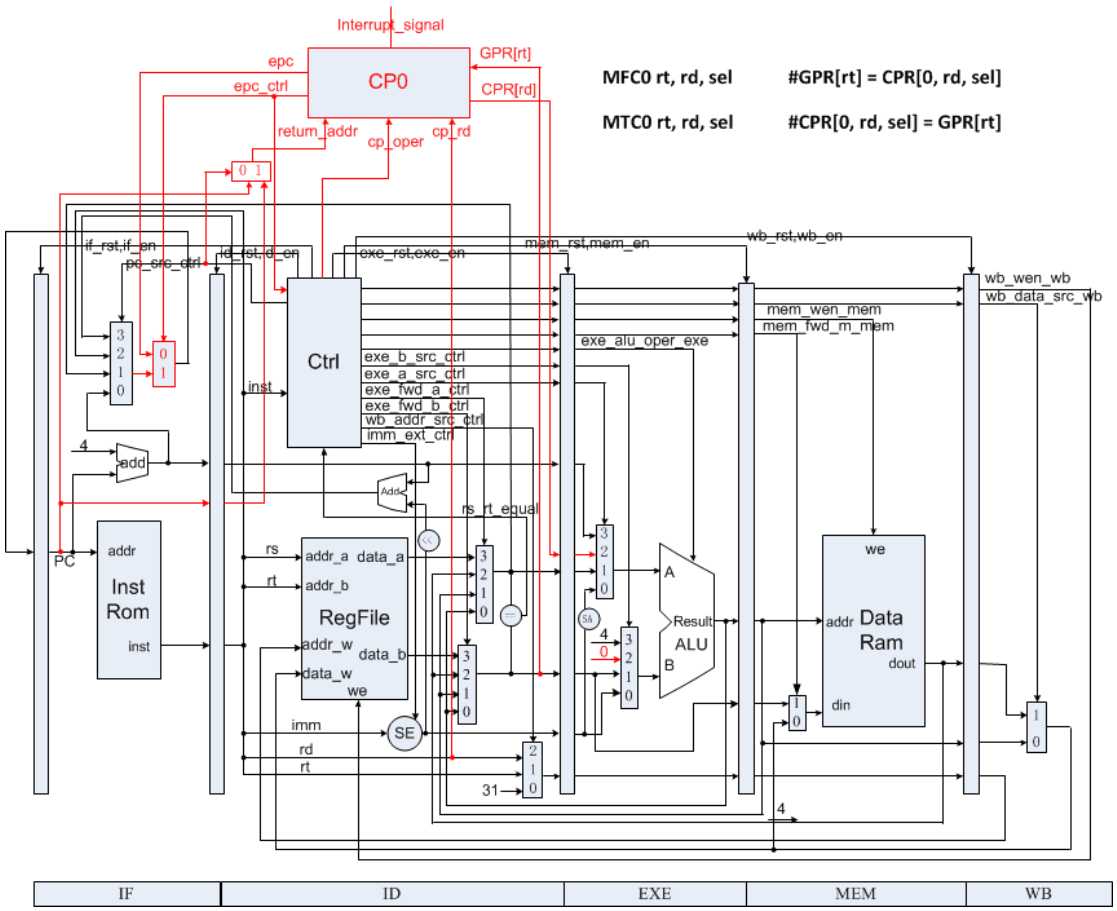
这层逻辑的实现也很简单，我们保存这次的 ir_in 和上个周期的 ir_in（ir_in_prev），如果这个周期的 ir_in 和 ir_in_prev 都为 1，也就是上个周期和这个周期都有中断信号产生，那么我们都不会产生 ir_in 信号，ir_in = ~ir_in_prev & ir_in，仅当这个周期有感知到按钮按压，而上个周期没有时，我们才考虑有中断信号出现。

参考代码如下，这里用 interrupt 信号代替了我们上文中所言的 ir_in 信号。

```
// interrupt
wire interrupt;
reg interrupt_prev;
always @(posedge clk) begin
    interrupt_prev <= interrupter;
end
assign interrupt = ~interrupt_prev & interrupter;
```

注意虽然这个逻辑处理非常类似，但是还是要注意，防抖动和防中断重进入是两件事，不要将概念混同，但是在实现是，确实可以将代码交叠在一起写。（我们也确实是这么干的）。

2.6 实验参考电路图：



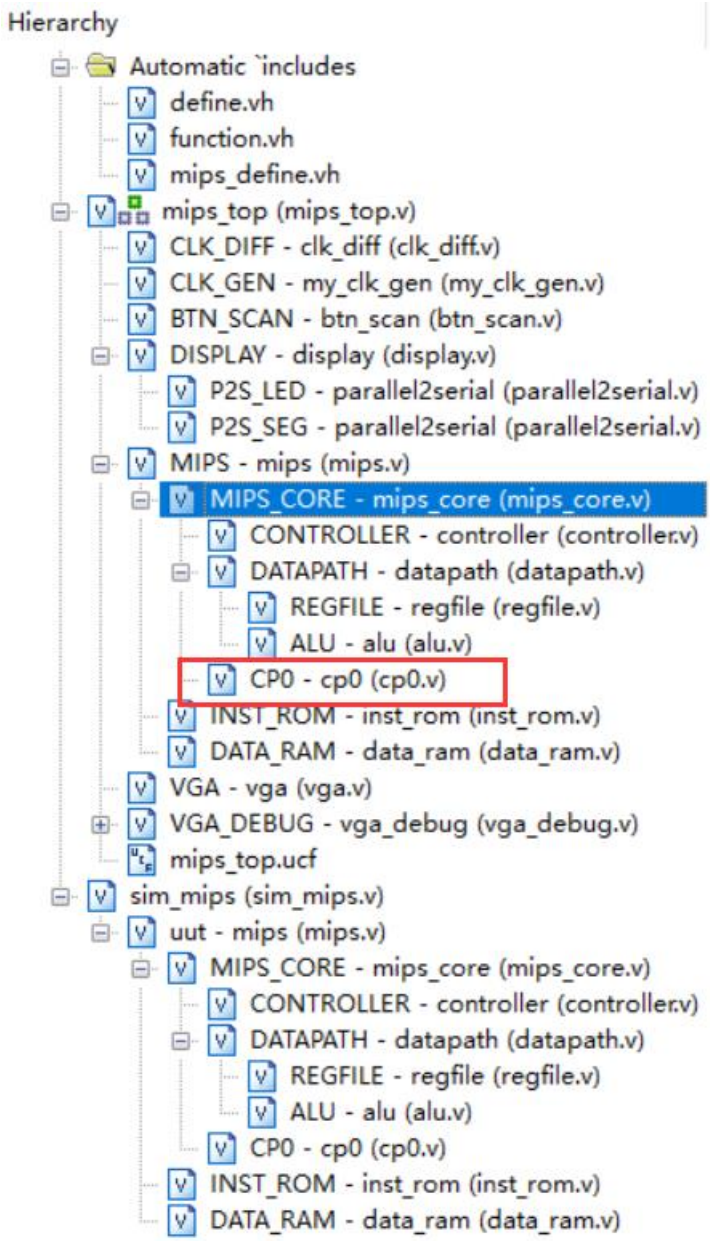
新增加的信号一进入前所言，这里不再赘述

2.7 我们的主要工作：

1. 处理 CPU Controller，添加 ERET，MFC0，MTC0 等信号的解码
2. 处理 CPU Datapath 模块，支持 ERET，MFC0，MTC0 的实现
3. 新增 CP0 模块，完整 ir 感知处理，以及 ERET，MFC0，MTC0 信号的处理
4. 处理 mips_core.v，将 Controller 的输出和 CP0，datapath 的输入连接起来（datapath 倒不需要做什么改动）
5. 对于新增信号的判断，我们如果想要引入宏增强可读性，那么我们需要修改 mips_define.vh
6. 我们需要处理 data_mem.hex 和 inst_mem.hex，修改测试程序，使得我们的新的指令能被测试到。

三、 实验过程和数据记录

实验的文件结构如下：



Automatic includes 文件夹下定义了我们可以在代码中使用的头文件；

mips_top.v 是实验的主要工程，实现了对于 MIPS 指令的模拟，clk_diff 和 clk_gen 模块用于 CPU 内置时钟的信号生成的处理，便于各个进程的同步。Btn_san 是处理输入信号的模块，时限完成的 mips_top 模块将烧录到实验室的 sword 开发板上，通过按钮的控制反映 CPU 的计算情况，信息通过 3 个模块输出，分别是 DISPLAY 模块和 VGA，VGA_DEBUG 模块，mips_top.ucf 规定了电路的引脚，使得烧录之后的程序可以嵌入开发板；

sim_mi.v 实际上是一个测试用的工程文件，用于在开发者的电脑上，用 Xilinx ISE 仿真烧录之后的输入输出过程，uut 文件指定了我们的模拟方式（通过内置时钟不断读取 inst_rom

和 data_ram 中的信息，并且把相关的信号反馈到电路图上），仿真检测的对象自然是 mips_core，这个 mips_core 核上文中 mips_top 模块中的 mips_core 是同一个文件。

我们需要完成 mips_core 中的代码补全。mips.v 文件划分为 MIPS_CORE 和 inst_rom，data_ram，完成了二者（CPU 和内存）的交互，mips_core.v 划分为 controller.v 和 datapath.v，完成了二者（控制器和数据通路）的交互，我们重点需要完成 controller.v 和 datapath.v 的代码补全。同时注意 cp0 的代码

1. CP0 和 datapath：

我们按照上文所述完整 CP0 的书写：

```
module cp0(
        input wire clk, //main clock
        //operation (reads in ID stage and write in EXE stage)
        input wire [1:0] oper, //CP0 operation type
        input wire [4:0] addr_r, //read address
        output wire [31:0] data_r,//read data, CPR[rd]
        input wire [4:0] addr_w, //write address
        input wire [31:0] data_w, //write data, CPR[rt]
        //exceptions (check exceptions in MEM stage)
        input wire rst, //synchronous reset
        input wire ir_en, //interrupt enable
        input wire ir_in, //external interrupt input
        input wire [31:0] ret_addr, // target instruction address to st
ore when interrupt occurred
        output reg jump_en, //force jump enable signal when interrupt a
uthorised or ERET occured
        output reg [31:0] jump_addr, //target instruction address to ju
mp to
        output reg ir,
        output reg ir_valid,
        output reg ir_wait
     );

    `include "mips_define.vh"

    reg [31:0] regs[31:0];


    //interrupt determination
    reg ir_in_prev = 0;
    wire eret;
    assign eret = (oper ==  EXE_CP0_ERET);
```

```
always @(posedge clk)begin
    if(rst)begin
        ir_valid = 1;
        ir_wait = 0;
        ir_in_prev = 0;
    end
    else begin
        if(eret)
            ir_valid = 1;
        else if (ir)
            ir_valid = 0;//prevent exception reenter
        if (ir_in && !ir_in_prev)
            ir_wait = 1;
        else if(eret)
            ir_wait = 0;
        ir_in_prev = ir_in;

    end
    ir = ir_en & ir_wait & ir_valid;
end

//Exception Handler Base Register
//Exception Program Counter Register
assign data_r = regs[addr_r];

//jump determination
always @(negedge clk)begin
    if(oper == EXE_CP0_ERET) begin //eret
        jump_addr = regs[CP0_EPCR];
        jump_en = 1;
    end
    else if (oper == EXE_CP_STORE) begin
        regs[addr_w] = data_w;
    end
    else if (ir) begin //external interrupt handling
        jump_addr = regs[CP0_EHBR];
        regs[CP0_EPCR] = ret_addr;
        jump_en = 1;
    end
    else begin
        jump_en = 0;
        jump_addr = 32'b0;
    end
```

```
    end
endmodule
```

其中 EXE_CP_STOR 表示的宏是 MTC0 被调用，此时我们需要写入数据，而读出数据的 MFC0 我们首先依靠 Datapath 实现索引计算，如上图所示，我们只需要在 ALU 部件的源中添加入口即可：

```
    always @(*) begin //addded EXE_A/B_IR; in chap 6
        opa_exe = fwda_exe;
        opb_exe = fwdb_exe;
        case (exe_a_src_exe)//0-1
            EXE_A_RS: opa_exe = fwda_exe;//
            EXE_A_NEXT: opa_exe = inst_addr_next_exe;//
            EXE_A_SA: opa_exe={27'b0, inst_data_exe[10:6]};//?
            EXE_A_IR: opa_exe = cp0_data_r_exe;////
            //EXE_A_BRANCH: opa_exe = inst_addr_next_exe;
            default:;
        endcase
        case (exe_b_src_exe)//0-2
            EXE_B_IMM: opb_exe = data_imm_exe;//
            EXE_B_FOUR: opb_exe = 3'h4;  //
            EXE_B_RT: opb_exe = fwdb_exe;//
            EXE_B_IR: opb_exe = 0;////
            //EXE_B_BRANCH: opb_exe = {data_imm_exe[29:0], 2'b0};
        endcase
```

ID 阶段我们也需要完成 MFC0、MTC0、ERET 的解码，datapath 应该有：

```
    assign
        //MTC0
        data_w = fwdb_id,//GPR[rt] in graph
        addr_w = inst_data_id[15:11],//cp_rd in graph

        //MFC0
        addr_r = inst_data_id[15:11],

        ir_en = 1,
        ret_addr = pc_src_ctrl ? inst_addr_id : inst_addr;//new multipl
exer for cp0
```

2. CPU Controller 和 mips_define

我们对于 controller 的内容修改如下，我们需要在解码时判断指令的操作码（opcode）是否和我们预定义的宏一致，如果是，我们需要根据不同的宏给数据通路发送信号，这些信号有时亦通过宏表示，为此我们需要修改 mips_define 来解释这些宏

```
case (inst[31:26])
    INST_R: begin
        case (inst[5:0])
                    . . . . . .
                      //Exp6 interrupt ----------------------
            INST_CP0: begin
                if(inst[25]) begin
                    if(inst[5:0] == CP0_CO_ERET) begin
                        oper = EXE_CP0_ERET;
                    end
                end
                else begin
                    case(inst[24:21])
                        CP_FUNC_MT: begin //MTC0
                            oper = EXE_CP_STORE;
                        end
                        CP_FUNC_MF:begin //MFC0
                            exe_a_src = EXE_A_IR;
                            exe_b_src = EXE_B_IR;
                            exe_alu_oper =  EXE_ALU_ADD;
                            wb_addr_src = WB_ADDR_RT;
                            wb_data_src = WB_DATA_ALU;
                            wb_wen = 1;
                        end
                    endcase
                end
            end
            //-------------------------------------
            default: begin
                 unrecognized = 1;
            end
      endcase
```

上文展示了所有新增加的CP0代码，我们使用宏来表示比较，这些宏定义在mips_include里面，展开之后是操作符对应的二进制码。

```
//CP0 registers
localparam
    //CP0_SR = 0,
    //CP0_EAR = 1,
    CP0_EPCR = 2,
    CP0_EHBR = 3;
    //CP0_IER = 4,
    //CP0_ICR = 5,
    //CP0_PDBR = 6,
```

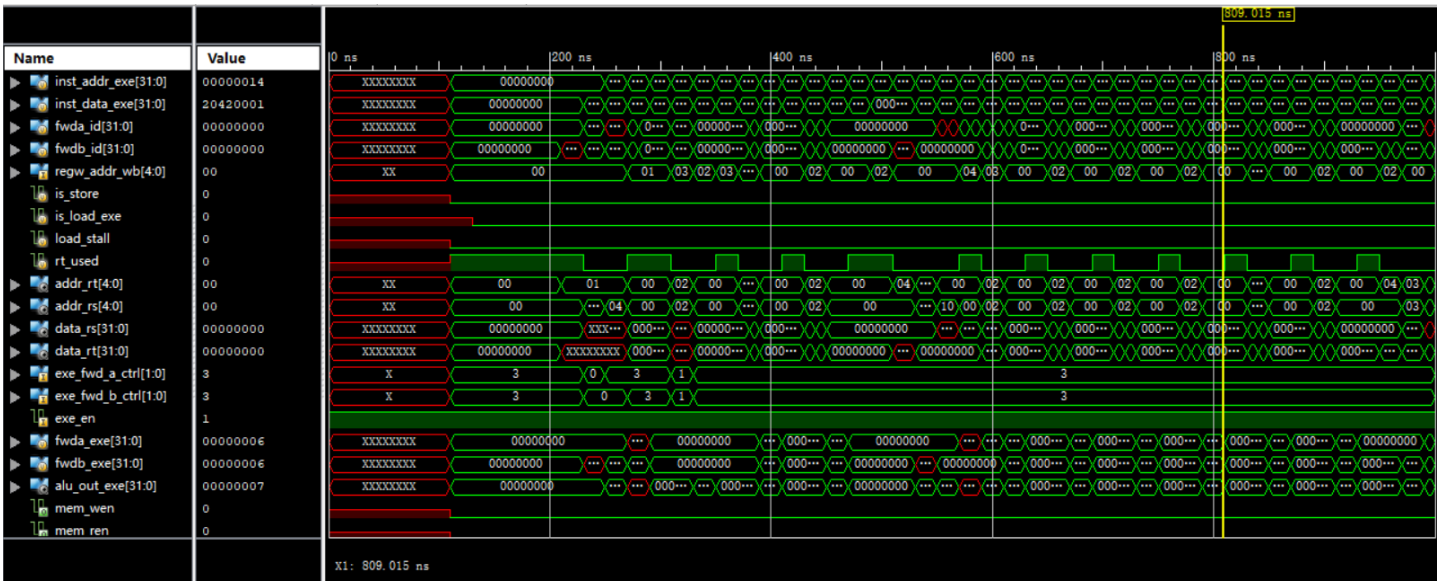
```
    //CP0_TIR = 7,
    //CP0_WDR = 8;

// EXE CP operations
localparam
    EXE_CP_NONE = 0,
    EXE_CP_STORE = 1,
    EXE_CP0_ERET = 2;
```

3. mips_core：

我们只需要按图进行结合就可以了，这里就不再赘述了。

四、 实验结果分析

4.1 仿真结果：



我们在图片右侧的紫框中找到相应的芯片元件，拖入左侧的图中就可以得到输入输出的信号波形图，注意设定仿真时长，并且显示方式为 16 进制(便于阅读)。

主要观察的波形是 regfile 和 inst 信号的输出，通过观察 inst 信号我们可以确定跳转的指令是否正确，通过观察 regfile 我们可以确定写回的信号是否正确。

4.2 测试用程序介绍

我们测试用的程序如下（注意和之前的程序不同）：

```
 0: 3c010000    lui     R1,0x0      # main entry
 4: 24210020    addiu   R1,R1,32
 8: 40811800    mtc0    R1, R3
 c: 00001020    add     R2,R0,R0
10: 00001820    add     R3,R0,R0
```

```
14: 20420001    addi    R2,R2,1     # loop
18: 08000005    j       14
1c: 00000000    nop
20: 40041000    mfc0    R4,R2       # handler
24: 20630001    addi    R3,R3,1
28: 42000018    eret
2c: 00000000    nop
```

最右侧的 16 进制数是机器码，直接存放在 inst_rom.hex 中，被 CPU 读取。左侧的内容是机器码翻译之后的汇编语言，以及汇编语言的语义（用注释表示）

本次实验不涉及到内存，所以我们不涉及 data_rom.hex，事实上，这个文件里面的内容可以是空的，也尅是任意的其他内容。

上述汇编语言的结果如下：首先，我们使用 lui 指令，将 R1 用 0 填充，再将 32 用 addiu 指令填入 R1（十六进制表现为 20），之后，我们把 R1 中的值移入到 CP0 的 R3 寄存器，之后，我们将 R2、R3 置 0，然后开始反复的 R2 加一的循环，知道我们外部加入一个终端，跳转到 EHBR 所在的地址，在我们的实现中，EHBR 对应的 CP0 寄存器是 R3，而因为 4 号指令的原因，CP0 寄存器 R3 已经存入了 20，于是我们将会跳转到 20 号指令 mfc0 R4 R2，将 CP0 寄存器的 R2 值读入通用寄存器的 R4，R2 按照我们的设计应当是 EPCR，就是产生中断的地址值，课件他应该是 14 或者 18，在这个循环体中中断的话，之后我们给 R3+1，就是说每发生一次中断，R3 的寄存器都应该加 1，然后 ERET，返回触发中断的地址。

4.3 仿真程序和测试结果：

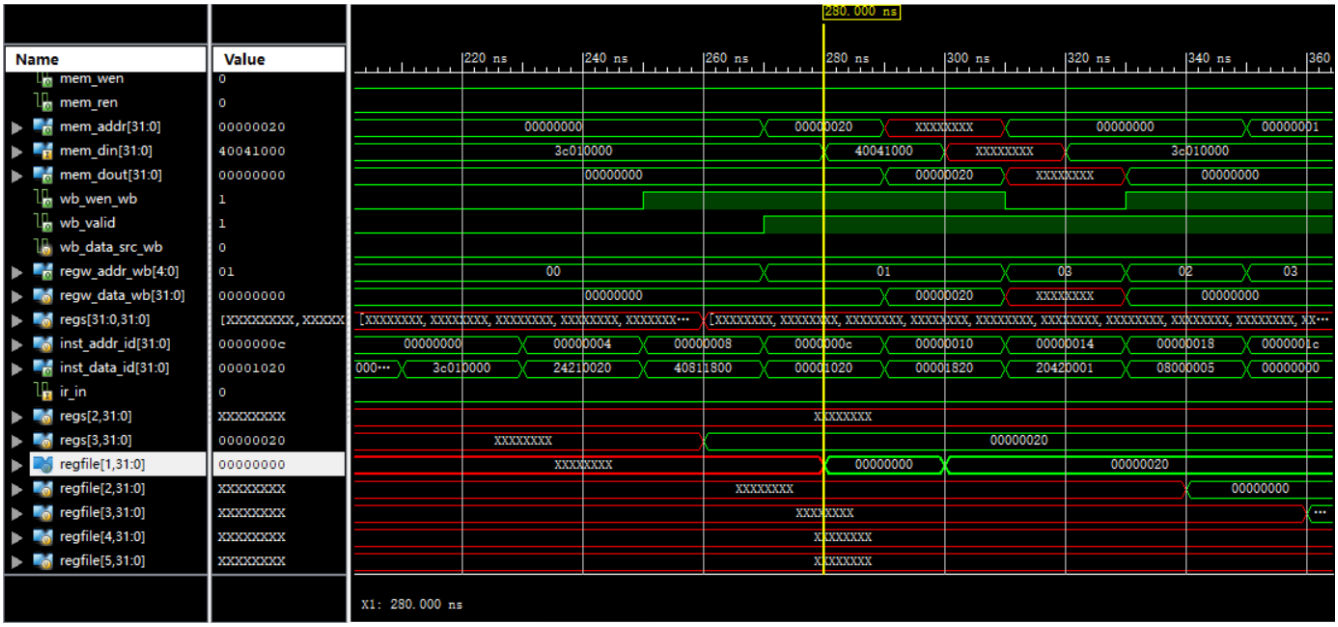
我们的仿真代码如下：

```
`timescale 1ns / 1ps


module sim_mips;

    // Inputs
    reg debug_en;
    reg debug_step;
    reg [6:0] debug_addr;
    reg clk;
    reg rst;
```

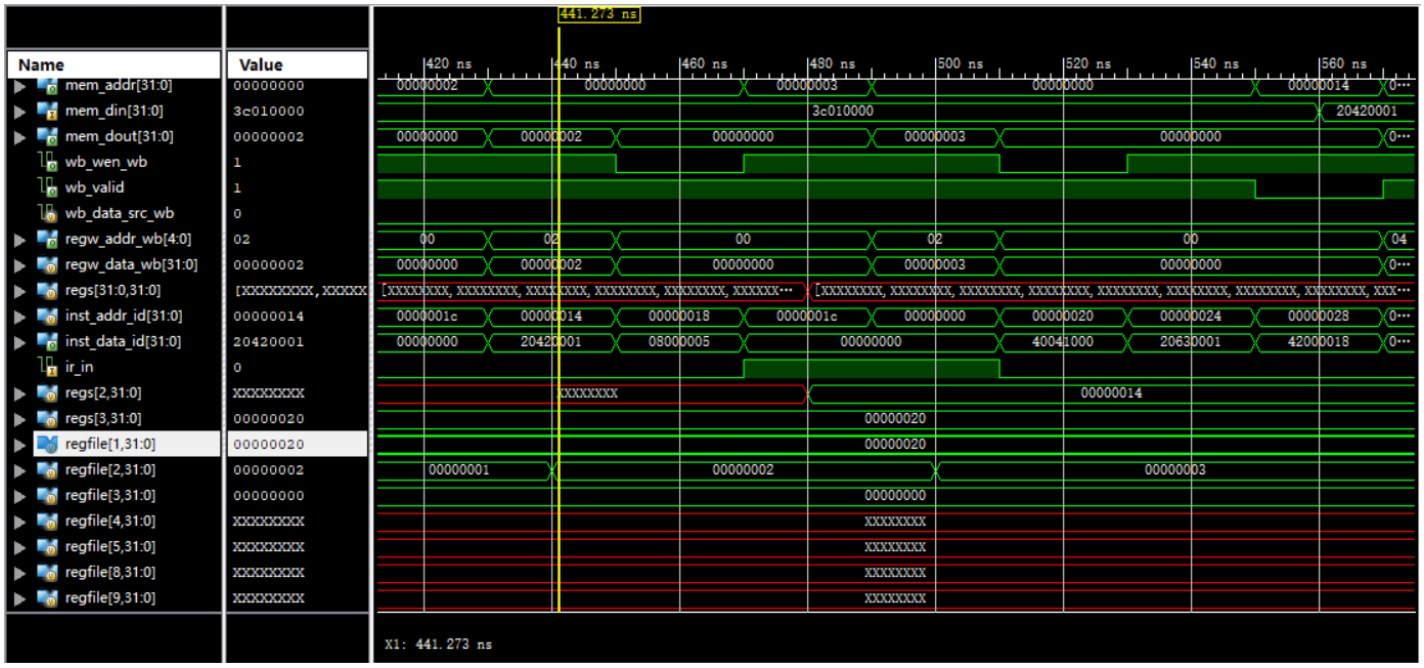
```
    reg interrupter;

    // Outputs
    wire [31:0] debug_data;

    // Instantiate the Unit Under Test (UUT)
    mips uut (
        .debug_en(debug_en),
        .debug_step(debug_step),
        .debug_addr(debug_addr),
        .debug_data(debug_data),
        .clk(clk),
        .rst(rst),
        .interrupter(interrupter)
    );

    initial begin
        // Initialize Inputs
        debug_en = 0;
        debug_step = 0;
        debug_addr = 0;
        clk = 0;
        rst = 0;
        interrupter = 0;

        #100 rst = 1;
        #100 rst = 0;

        #270; interrupter = 1;
        #40; interrupter = 0;

        #400 interrupter = 1;
        #400 interrupter = 0;

        #180;
    end
    initial forever #10 clk = ~clk;

endmodule
```

可见，我们在 470ns 设置了一次中断，又在 910ns 时重新设置了一次。

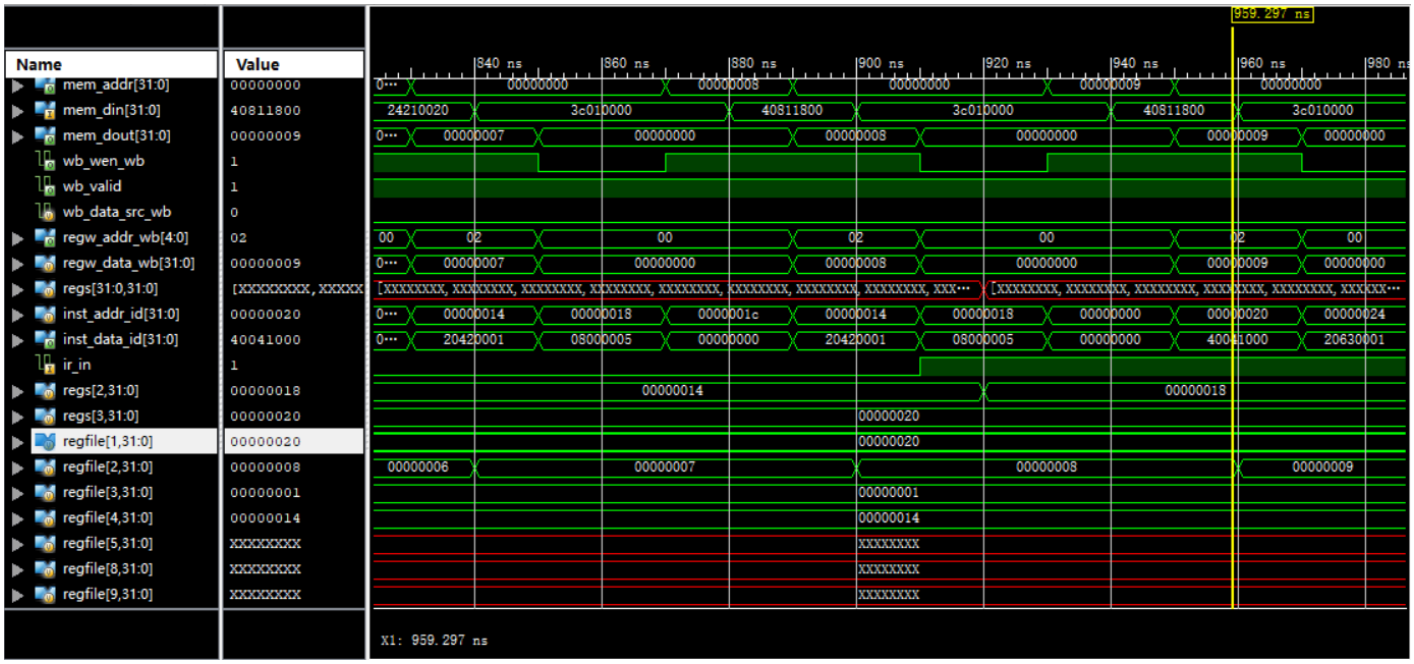
开始阶段：



（第一次中断部分）



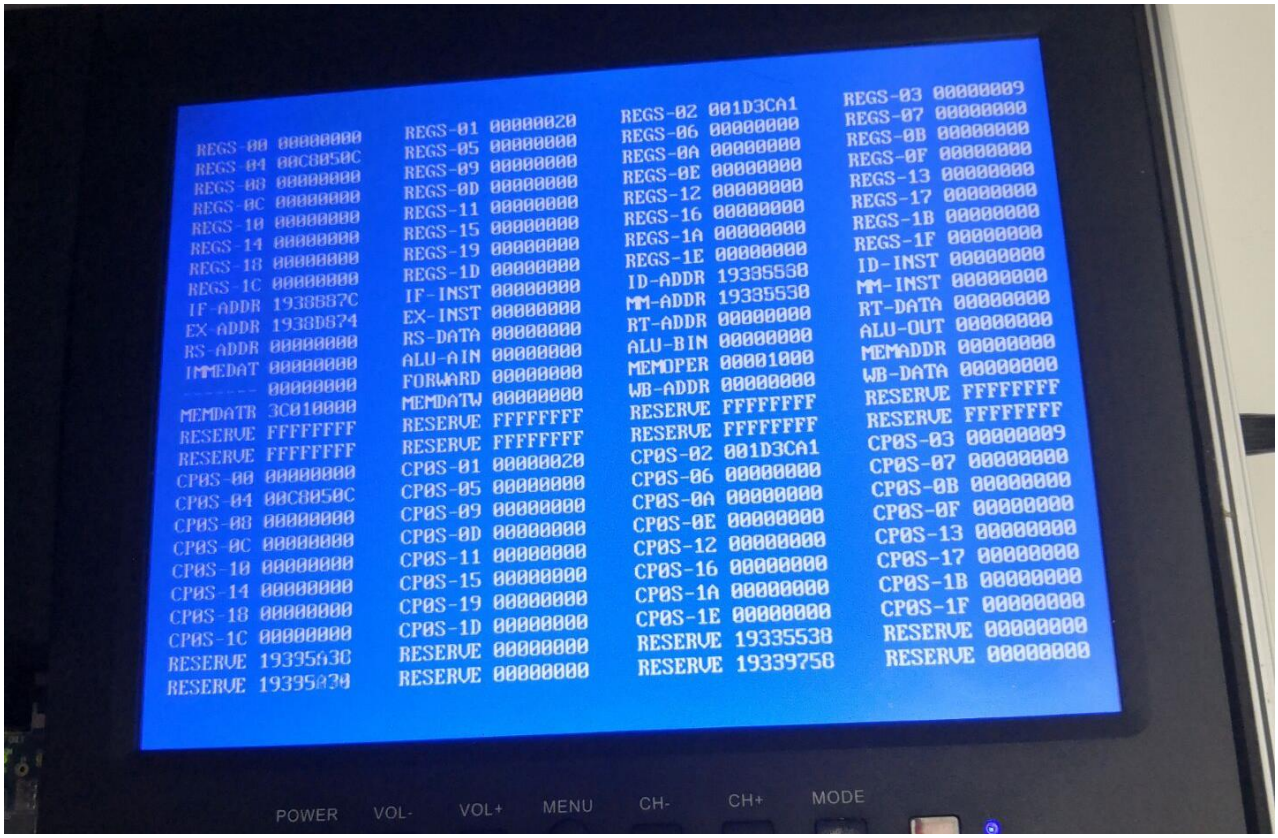
（第二次中断循环部分）



4.4 在 sword 开发板上的仿真结果

因为已经通过现场测试，这里就不再展示全部的情况，仅仅选择部分展示

下文节选一些中间过程



实验中 mips_top.v 中的语句

```
    assign
```

```
        btn_step = btn[16],
        btn_interrupt = btn[19];////interrupt button
```

将触发中断的信号绑定在了 19 号按钮上，我们可以按压开发板上的这一按钮实现中断。

五、 讨论与心得

实验的主要难点在于理解中断信号接收部分的处理，那些 ir_in，ir_valid, ir_wait 的存在确实令人比较迷惑，但是最后我们还是获得了比较令人满意的理解。